Using a Multifrontal Sparse Solver in a High Performance Finite Element Code

SCOTT D. KING

Division of Geological and Planetary Sciences,

California Institute of Technology, Pasadena, CA 91125

ROBERT LUCAS

Supercomputer Research Center

19200 Science Dr., Bowie, MD 20716

ARTHUR RAEFSKY

Electrical Engineering Dept.

Stanford University, Stanford, CA 94305

Proposed running title: Multifrontal Sparse Solver

Address correspondence to:

Scott King

IGPP 0225

University of California at San Diego

La Jolla, CA 92093-0225

We consider the performance of the finite element method on a vector supercomputer. The computationally intensive parts of the finite element method are typically the individual element forms and the solution of the global stiffness matrix both of which are vectorized in high performance codes. To further increase throughput, new algorithms are needed. We compare a multifrontal sparse solver to a traditional skyline solver in a finite element code on a vector supercomputer. The multifrontal solver uses the Multiple-Minimum Degree reordering heuristic to reduce the number of operations required to factor a sparse matrix and full matrix computational kernels (e.g. BLAS3) to enhance vector performance. The net result is an order-of-magnitude reduction in run time for a finite element application on one processor of a Cray X-MP.

Introduction

The finite element method is a powerful tool for solving differential equations arising in: continuum mechanics, structural analysis, and computational fluid dynamics (CFD) as well as many other applications. Because these problems are so large and time consuming, it is only natural to use the largest and fastest computers available to solve them. Today this means using vector machines like the Cray X-MP. In a previous paper (King *et al.*, 1990), a strategy for vectorizing the finite element method was discussed in the context of ConMan, a geophysical CFD program. This paper builds upon this work, describing the incorporation of a multifrontal sparse solver into ConMan.

Any finite element algorithm consists of two main computational tasks: forming the global matrix and then solving this large sparse system. For high performance both the element forms and the sparse solver need to be optimized. Because of the flexibility of the finite element method, both in the arbitrary nature of the domain and grid and the flexibility of the boundary conditions, the process of forming and assembling the global matrix equation is more computationally intensive than most finite difference or spectral methods. In fact, some problems (e.g., explicit methods and algorithms for non-linear problems) are dominated by the formation of the matrix, not the solution.

Usually however, the major amount of computation in a finite element code is in the solution of the large, sparse, global matrix. The matrices resulting from finite element discretizations are often ill-conditioned. The reasons for this include the non-uniform structure of the grid and the enforcement of constraints such as incompressibility. There-

fore, in general iterative solvers do not perform well on these problems and direct methods tend to be used instead. Historically, these have been band or skyline solvers (Zienkiewicz, 1977) that are relatively easily vectorized. General sparse solvers (George and Liu, 1981) were not used because, while they often performed less arithmetic operations and required less storage, they didn't vectorize well. However, over the course of the last decade, multifrontal (Duff, 1986; Ashcraft, 1987) and supernodal general sparse methods (Ashcraft, 1987; Simon *et al.*, 1989) have been developed that combine asymptotic vector computational speed (i.e., MFLOPS) with minimal arithmetic operations. As this paper shows, incorporation of one of these sparse solvers has a dramatic effect upon the performance of a finite element application.

For completeness, this paper begins in Section 1 with a discussion of vectorization modifications to the element form routines. It then describes the competing sparse matrix solvers in Section 2. Opportunities for exploiting parallel aspects of the element forms and the multifrontal sparse solver will be mentioned. Section 3 will then demonstrate the enhanced performance of ConMan on single CPU vector machines. Computational speeds approaching half the theoretical machine limit for the entire code, including I/O and subroutine overheads, is demonstrated using standard Fortran, without hand coding routines in assembly language. Finally, conclusions and suggestions for future work are presented in Section 4.

Results and Discussion

1. Finite Element Assembly

The process of assembling the global matrix equation, called the stiffness matrix, involves three tasks; localizing the data for each element from global data arrays, forming the element stiffness matrix, and then assembling the element stiffness matrix into the right coefficients in the global stiffness matrix. This is done using indirect referencing, where an array called the *lm* or "location matrix" is used to gather the global data into local work arrays and scatter the element stiffness matrix into the global stiffness matrix. The *lm* array takes the element number and the local node number as arguments and its value is the corresponding equation number in the global stiffness matrix. Table 1 shows the values of the *lm* array for a simple grid of bilinear elements with one degree of freedom per node as shown in Figure 1.

Most finite element programs use an element level algorithm; all operations are performed for a given element which is then assembled into the global matrix equation. This approach does not work well on vector machines because there are very few vector operations at the element level and the vector lengths are short. Fortunately, forming the element equations involves repeating the same operations for each element and there are typically thousands of elements in a finite element problem. Furthermore, each element is independent of the others. Therefore, one can take advantage of the parallel nature of the element forms by performing the operations which would normally be done of a single

element on a block of independent elements using vector arithmetic operations that span the entire block. This requires slight modifications to the data structures but no major changes to the algorithms.

Assembly of the element stiffness matrices into the global matrix equation poses a potential problem when processing the elements in vectors. Consider bilinear elements with one degree of freedom per node as in Figure 1. When assembling the equation for global degree of freedom 5, there is a contribution from elements 1, 2, 3 and 4 (element identifiers are circled). If any of these four elements are grouped in the same vector block, then the value of the global matrix for equation 5 may be updated incorrectly. This is an example of a simple vector dependency. To prevent it, one needs to make sure that no coefficient in the global stiffness matrix is updated twice from within the same vector. Independence can be assured by requiring that no element in a vector share nodes with any other element in the same vector. For bilinear quadrilateral elements, this is accomplished by separating the elements into groups using a "four-color" scheme as shown in Figure 2. The coloring scheme can be easily extended to more complicated grids containing mixtures of triangles and quadrilaterals as well as higher dimensional grids.

In practice, collecting elements into these blocks can be implemented by reordering the *lm* array, which need only be performed once for each grid used and can be done as part of the initial set up. In the element form routines there are only a few minor changes required to avoid vector dependencies. These are illustrated by the two fragments

of Fortran code in Figure 3. First loops over elements are replaced by two loops, an outer loop over element blocks and an inner loop over elements within the block. The inner loop can be safely vectorized because of the reordering of *lm*. Also, loops over the local element degrees of freedom are unrolled since they would otherwise be too short to achieve asymptotic vector speed (typically length 4 or 8 for bilinear quadrilateral elements see, Figure 3).

The amount of storage needed to form a block of elements is directly related to the size of the block. Only a few blocks are necessary to avoid vector dependencies, and as this storage is transient, it is useful to further subdivide the blocks for most problems. For register based machines (e.g., Cray, Convex, Alliant, Stardent), the ideal block size is the vector register length. This temporary storage is insignificant compared to the size of the global matrix.

Finally, the blocks of elements can be assembled by different processors as long as interprocessor write dependencies are prevented. This can be accomplished without undo synchronization overheads by having the processors accumulate their coefficients into independent subsets of the global stiffness matrix. This restriction is enforced on hypercubes by the independent address spaces provided for each processor. Lyzenga, Raefsky and Nour-Omid (1988) have demonstrated speed-ups close to the peak parallel efficiency of such machines using this method. Note that coloring can be done within each of the domains for parallel vector implementations.

2. Factorization

Usually the solution of the global matrix dominates the computation in a finite element code. Iterative solvers like preconditioned conjugate gradients or multigrid can be the fastest methods for solving these problems provided that the number of iterations needed for the solution to converge is small. Iterative solvers are also the most efficient in terms of storage. Because of this much attention has been given to using iterative methods for finite element codes (Ferencz, 1989).

Unfortunately, in many finite element methods the stiffness matrices are ill-conditioned. As a result, the number of iterations required for convergence is so large that often direct solvers are used instead of iterative solvers even for many three-dimensional applications (Hallquist, 1984). The following subsections will discuss first the traditional band oriented direct sparse solvers, then reorderings which reduce the work performed, and finally the multifrontal solver.

2.1 Traditional Sparse Solvers

Traditionally, a band or skyline type algorithm is used in vectorized finite element applications. There are many variations. The half bandwidth of a matrix, β, is defined to be the maximum $|i-j|$ such that a coefficient of the matrix $A(i,j)$ is not equal to zero. All coefficients within the band (e.g., $A(i,j): \quad |i-j| \leq \beta$) are assumed to be nonzero while those outside are zero. This extremely simple structure permits one to easily implement very fast vectorized computational kernels based upon outer products.

All vector operations can be performed with a stride of one and the vector lengths equal β. As a result, band solvers achieve speeds approaching the asymptotic peak of a vector processor.

One disadvantage of band solvers is that the bandwidth is determined by the maximum distance from the diagonal of a nonzero coefficient. Should this not be representative of the width of other rows, then many zeros will be incorporated within the band, adding unnecessary storage and operations. Envelope methods reduce these overheads by locally varying the bandwidth to better fit the nonzero structure of the matrix. Of course, this requires somewhat more complicated data structures to implement.

Often, operations can be further reduced by using skyline solvers. Each column in the upper triangle is assumed to fill in between its first nonzero entry and the diagonal as shown in Figure 4. Rows in the lower triangle fill in between their leftmost nonzero coefficient and the diagonal. While this reduces storage and operation counts, it doesn't come without a price. Updates to a column generated by the elimination of a previous column are accumulated with inner-products, $a = a + \sum_{i=1}^{n} b(i) * c(i)$. This requires a vector reduction which cannot run at peak rates on high performance pipelined machines where the floating point adders have multiple pipeline stages.

Figure 5 contrasts the above sparse solvers. Suppose one wishes to factor a matrix with the structure shown in Figure 5a. A strict band solver would fill in and operate on all elements within the shaded area of in Figure 5b. The envelope solver would preserve the zero structure in the center of the band as depicted in Figure 5c. The skyline solver

would require the least storage as shown in Figure 5d. It would maintain sparsity in the upper diagonal block as well as the center of the band. It only suffers fill in the lower diagonal block.

2.2 Sparse Matrix Reorderings

Both the band and skyline algorithms have the property that they constrain the work need to factor a sparse matrix to something substantially below the $O(N^3)$ of a dense factorization (where N is the number of equations). If one takes as a model problem a square grid, with n nodes on a side ($n = \sqrt{N}$), then it is easy to show that the operation count of a band or skyline factorization of the corresponding sparse matrix would be $O(n^4)$. If the grid has a more complicated structure, heuristics such as Reverse Cuthill-McKee (RCM) (George, 1971) or Gibbs-Poole-Stockmeyer (GPS) (Gibbs *et al.*, 1976) can be used to reorder the equations in such a way as to minimize the bandwidth or profile (i.e., number of nonzeros in a skyline algorithm) and thus the cost of the factorization.

However, one can usually reduce the operation count of a sparse matrix factorization even further. For the above mentioned model problem the Nested Dissection (ND) algorithm (George, 1973) splits the grid into two disjoint halves by removing a column of vertices from the center of the square. These vertices, called a separator, correspond to the last equations eliminated. The effect is depicted in Figure 6 where the vertical separator corresponds to the dense block at the lower right-hand side of the matrix. The remaining diagonal blocks are disconnected. The dissection process is then applied re-

cursively to the two halves of the grid. The resulting sparse matrix can be factored with $O(n^3)$ operations. Unfortunately, for more complicated grids it becomes very difficult to find acceptable separators.

Another heuristic that substantially reduces the operations needed to factor a sparse matrix is the Multiple Minimum Degree (MMD) algorithm (George and Liu, 1987). It eliminates an independent set of equations of minimum degree (i.e., lowest number of off-diagonal nonzeros), updates the remaining matrix, and then repeats the process. For the model problem, the operation count is close to that of the ND algorithm. Furthermore, it is based on a local optimization (minimum degree) that obviates any need for global knowledge of the structure of the grid. As a result, it performs well on unstructured grids as well and has become the reordering of choice for many sparse matrix codes. Implementations of the MMD heuristic are now available that reorder a sparse matrix in roughly the time it takes to perform one factorization. For transient non-linear problems, where the same grid is used many times, this overhead is quite reasonable.

One way of expressing the results of any of the above reorderings is through the use of an elimination precedence graph or elimination tree (Schrieber, 1982). The root of the elimination tree is the last equation in the matrix to be eliminated. Its children in the tree must be eliminated before it is. They are independent of one another and may be processed in any order, even concurrently. Figure 7 shows the elimination tree for a 5 by 5 grid reordered using the ND algorithm. The Nested Dissection algorithm generates the tree by finding the root first and the leaves last. For this small problem, the MMD

algorithm generates the same ordering, only it finds the leaves of the tree first and the root last. Both the ND and MMD algorithms have the effect of scattering non-zeros away from the diagonal and generating short, bushy elimination trees. In the process, they break up the simple, unit stride vectors that allow the band and skyline algorithms to achieve high computational speed. In contrast, the elimination tree for a minimum bandwidth ordering, in this case the natural ordering, would simply be a line, starting at 1 and ending at 25.

2.3 Multifrontal Sparse Solver

The multifrontal algorithm attempts to combine the best of two seeming incompatible goals. First, it allows an application to use a reordering such as MMD, which minimizes the work required to factor a matrix. Arithmetic operations are then performed with dense matrix arithmetic kernels that achieve very high vector processing speeds. The next four paragraphs will describe the multifrontal algorithm. It will then be illustrated with an example.

The multifrontal algorithm selects the order in which to eliminate the equations by performing a post-order traversal of the elimination tree. Starting from the root, we push each equation onto a stack and defer its elimination until its children have been processed. When a leaf in the tree is encountered, it can be eliminated immediately. The equations in Figure 7 could be eliminated in the following order: 1 2 6 7 5 4 10 9 3 8 21 22 16 17 25 24 20 19 23 18 11 12 13 14 15.

To eliminate an equation, it is collected into a small dense matrix called the front. First the front is cleared. Then the initial values for the pivot equation are scattered into the front. These values are the diagonal, the nonzeros below the diagonal in the pivot column, and those in the pivot row above the diagonal. Finally, they are added to any updates generated when factoring the pivot equation's children in the elimination tree. These operations represent overhead associated with the multifrontal algorithm and are greatly assisted by the presence of hardware support for indirect addressing (i.e., gather/scatter), chaining, and multiple memory ports in a vector processor.

Once an equation's front has been assembled, the equation can be eliminated. The diagonal is inverted, the pivot column normalized and negated, and then the Schur complement computed by an outer product of the pivot row and the pivot column. Afterwards, the equation's factors are stored away, and its Schur complement placed upon a stack from which it can be retrieved when needed for the assembly of its parent's front. Note, storage of this "real stack" represents an additional overhead associated with the multifrontal method.

The above mentioned overheads are ameliorated by exploiting supernodes. A supernode is one or more equations whose nonzero structures are indistinguishable. If the equations in a supernode are eliminated together, then their Schur complement can be computed using a matrix-matrix product. Using loop unrolling and other simple optimizations, matrix-matrix products can be performed at near peak speeds on vector processors. Furthermore, the cost of assembling the front for the supernode is amortized

over the factorization of many equations, reducing the multifrontal algorithm's overhead. If the problem is big enough such that the cost of factoring the supernode is significantly greater than that of assembling it, then the multifrontal algorithm will achieve near peak computational speed on modern vector machines.

Figure 8 illustrates the multifrontal algorithm by detailing the elimination of 2 equations from a contrived system of 25 equations. The sparse matrix in Figure 8a corresponds to the 5 by 5 grid with a 5-point stencil shown in Figure 7a. Zero coefficients are left blank for clarity. The matrix has been permuted to reflect the ordering in Figure 7b. The original row and column numbers are displayed along the left side and the top. If equations 3 and 8 are grouped into a supernode and eliminated together, then Figure 8b depicts the front after it has been cleared and the initial values scattered into it. Figure 8c contains the real stack with the update matrices left by the elimination of equations 7 and 9. Zeros have been included where coefficients were cancelled to illustrate the structure of the sparse matrix. The fully assembled front is shown in Figure 8d. Figure 8e contains the contents of the front after equations 3 and 8 have been eliminated. Eliminating equations 3 and 8 together introduces two unnecessary zeros into the front (coefficients 3,13 and 13,3). However, the updates to equations 11-15 can then be computed with the product of 5x2 and 2x5 matrices. This tradeoff is called "relaxation of supernode partitions" by Ashcraft (Ashcraft and Grimes, 1987).

The element form routines described in Section 1 and the above mentioned multifrontal sparse solver are both highly optimized. If the exchange of data between them

isn't, then a scalar bottleneck can be introduced. If storage is not an issue, then this is easily avoided. For each coefficient of each element, the corresponding location in the global stiffness matrix is computed and stored in a mapping array whose size matches that of the element stiffness matrix. Prior to formation of the elements, the global stiffness matrix is cleared. Then, each time a block of elements is formed, the corresponding mapping arrays are used to gather the appropriate coefficients of the global stiffness matrix into a vector which is added to the newly computed values. The mapping array is then used to scatter the updated coefficients back into the global stiffness matrix. The entire operation runs at vector stream rates on machines with gather/scatter hardware.

One feature of the above interface is that the finite element application need not be cognizant of the structure of the reordered matrix. This information is ensconced in the mapping arrays. The mapping arrays need only be computed once for each grid and their generation is simply a small one time overhead akin to the matrix reordering.

Finally, there are two rather obvious places to exploit concurrency in the multifrontal algorithm. The first is in factorization of the dense frontal matrices. After each pivot equation (or group thereof) is eliminated, the remaining columns can be updated by multiple processors. This "column wrap" distribution of the updates has been performed successfully on hypercubes (Moler, 1986), where the programmer controls the distribution of data, and on shared memory machines, where modern compilers can dynamically distribute loops. The latter process is called "Autotasking" on Cray machines. The second place to exploit concurrency is by simultaneously assembling and factoring fronts

from multiple branches of the elimination tree. This has been the key to the success of concurrent sparse matrix factorization algorithms on hypercubes (Lucas *et al.*, 1987; George *et al.*, 1986; Ashcraft *et al.*, 1990) and has been applied to vector supercomputers as well (Duff and Reid, 1986).

3. ConMan: An Example from Geophysical CFD

ConMan (CONvective MANtle), the application used here to illustrate the speed-up arising from both vectorized element forms and the multifrontal solver is a finite element code used for studying convection in the Earth's mantle (King and Hager, 1989; King *et al.*, 1990). It is a 2-D, incompressible, creep-flow program that uses the penalty method to enforce incompressibility in the velocity equations and Streamline-Upwind Petrov-Galerkin weighting functions for the advection terms in the temperature equation. The time-stepping is done with a second order predictor-corrector algorithm so that the time evolution of the flow can be accurately followed. This code has been benchmarked against other Cartesian convection codes (Travis *et al.*, 1990) and is more fully described in King *et al.* (1990).

At the elevated temperatures in the Earth's interior, rocks respond to stress by slow creeping flow. Thermal convection is the driving mechanism for plate tectonics and is the dominant mechanism for heat transfer in the Earth. In order to simulate rigid plates and a ductile mantle, a temperature-dependent, non-linear viscosity is required. The variation in effective viscosity coupled with the enforcement of incompressibility leads to

ill-conditioned matrices. Therefore, a direct sparse matrix solver for ConMan is needed. For temperature-dependent problems the velocity matrix is reformed and factored every time-step. Thus, for these problems the sparse matrix factorization time dominates all other routines.

Results are presented contrasting the use of multifrontal and skyline solvers, performing symmetric factorizations. For the problems studied, both computational speed (i.e., MFLOP rates) as well as run time are presented. This illustrates the fact that the multifrontal algorithm not only does less work, it also achieves higher vector computational speed. Since users are typically charged for CPU time the run time speed-up is the more important result.

Table 2 shows the parameters for the grids used in the four benchmark problems. The first three grids are all square regular meshes with uniform size elements. The resulting stiffness matrices are banded. This provides a test of the algorithms as a function of an increasing number of operations. The last benchmark has a periodic boundary condition where fluid leaving the left side of the box reenters the right side. This gives a matrix structure similar to the pattern in Figure 4. BM4 is typical of the type of problems ConMan is actually used for in geophysical fluid dynamics. For this problem the multifrontal solver is noticeably better at reducing the storage and the amount of computations.

The execution times for the four problems on one processor of a Cray X-MP are shown in Table 3. The three columns correspond to the skyline version of ConMan

with and without vectorization enabled and the vectorized multifrontal version. They illustrate the improvements in computational speed gained both by the vectorization enhancements to the element forms as well as the addition of the multifrontal sparse solver. The total times for the three versions of the four benchmarks are summarized in Figure 9. The rows of Table 3 break out the time spent in various portions of the code. The routines *f_vstf, f_vres,* and *f_tres* are the finite element assemble and form routines for the velocity stiffness matrix, velocity right hand side and temperature right hand side respectively. The routines *factor* and *back* are for the matrix factorization and the forward-reduction/back-substitution. The routine *symbol* does the matrix reordering and symbolic factorization and is only used in the multifrontal code. The natural ordering is optimal for the skyline solver so no reordering is needed. Since the meshes do not change with time, the reordering for the multifrontal solver is only done once while the factorization and backsolves are done every time step. The itemized times in Table 3 do not add up to the total because there are other routines for the grid generation, output, etc. The results for BM4 show that for real problems the multifrontal solver performs an order of magnitude faster than the vectorized skyline solver. All these results are using Fortran compiled with the cft77 compiler. Using Cray's optimized scilib routine SMXPY in the multifrontal solver increases its performance another 15%.

Figure 10 shows how the element form times improve as a function of grid size. For BM1, BM2 and BM3, the operations in the element routines increase linearly with the total number of elements ($O(n^2)$ where n is the number of elements on a side in an

$n \times n$ mesh). This is reflected by the quadratic shape of the scalar time vs. n curve. The vector time grows much more slowly with increasing number of elements. This is due to the increasing efficiency of the vector functional units as the problem size and hence the vector lengths increase. There are more operations over which to amortize the cost of starting up vector operations.

Figure 11 shows the breakdown in routine *f_vstf* between the computation time and the time to assemble the global stiffness matrix. Assembling the global stiffness matrix is where the indirect addressing costs are the worst because the element matrices are being scattered into the sparse matrix structure. This represents the interface between the finite element assembly and the multifrontal solver. Figure 11 clearly shows that this is not a bottleneck.

Figure 12 depicts the time spent factoring the stiffness matrix for BM1, BM2 and BM3. It shows that like the element formations, the performance of the multifrontal solver improves with increasing grid size. In contrast, the skyline solver which is dominated by vector reductions shows the same trend as the scalar finite element routine in Figure 10. For the smaller grids, the full matrices factored in the multifrontal algorithm are small which corresponds to short, suboptimal vector lengths. For the larger problems, the average vector length is greater, increasing vector computational speed.

When comparing the scalar to multifrontal results, remember that there are three components of speed-up in the these results. The first is the increase in computational speed afforded by vectorizing the element formation routines. The second is the reduction

in operations performed by using the MMD reordering for the sparse solver. The third is the switch from the use of inner products in the skyline solver to outer products in the dense matrix computational kernels in the multifrontal solver. The total speed-up exceeds the theoretical scalar/vector speed-up (210/9.5 on the Cray X-MP) because the speed-up includes the reduced number of operations performed by the multifrontal solver.

Finally, note that the scalar times for the element routines (*f_tres, f_vres, f_vstf*) are greater than the time for the multifrontal sparse matrix factorization. If the only optimization were the new equation solver, then the run time would be dominated by the element formation, and only a marginal speed-up would be seen.

4. Conclusions

The multifrontal solver requires fewer operations and achieves higher vector computational speed than the skyline solver for the finite element problems considered. For large problems, the difference in CPU time required by the solvers is nearly a factor of twelve and the the entire application runs 9.5 times faster. The speedup perceived by the user is typically even greater. The multifrontal version of the code requires less storage and less CPU time so it can be run in higher priority scheduling queues. For three dimensional problems the difference between the skyline and multifrontal solver will be even larger.

Future work needs to proceed in two directions. First, there needs to be more work on heuristics for reordering unstructured grids. For 3-D rectangular grids, for which it is easy to contrast the MMD algorithm to the asymptotically optimal ND, the MMD

reordering can create two to three times the work of a ND reordering. Secondly, the use of multiple processors needs to be explored. Until more effective iterative solvers are devised, parallel processing affords the next obvious opportunity for improvement in the computational speed of finite element applications.

A final note: the original version of ConMan, run on a VAX 11/780 took about 8 hours of CPU time for BM1. Thus the total speed-up as perceived by the user, including improvements in both the algorithm and the hardware, is 450. It is important to remember that this is for the smallest of the sample problems. Larger ones, such as BM4, were simply not feasible.

Acknowledgements

We thank Horst Simon for encouraging us to design a simple and clean interface for the DMF solver. We also thank Greg Lyzenga for his helpful comments on this manuscript. This is contribution 4930 of the Division of Geological and Planetary Sciences, California Institute of Technology. Scott King was supported by NASA grant NAG5-1132 and NSF grant EAR-8618744. All benchmarks were executed on the Cray X-MP/18 at JPL.

References

Ashcraft, C. C., and Grimes, R. 1987. The influence of relaxed supernode partitions on the multifrontal method. Engineering Technology Applications Technical Report, ETA-TR-60. Seattle: Boeing Computer Services.

Ashcraft, C. C., Eisenstat, S., and Liu, J. 1990. A fan-in algorithm for distributed sparse matrix factorization. *SIAM J. Sci. Statist. Comput.* 11(3): 593-599.

Ashcraft, C. C. 1987. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. Engineering Technology Applications Technical Report, ETA-TR-51. Seattle: Boeing Computer Services.

Ashcraft, C. C., Grimes, R. G., Lewis, J. G., Peyton, B. W., and Simon H. D. 1987. Progress in sparse matrix methods for large linear systems on vector supercomputers *Internat. J. Supercomput. Appl.* 1(4):10-30.

Duff, I. S. 1986. Parallel implementation of multifrontal schemes. *Parallel Computing.* 3: 193-204.

Duff, I. S., and Reid, J. K. 1983. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software.* 9(3): 302 - 325.

Ferencz, R. M. 1989. Element by element preconditioning techneques for large scale vectorized finite element analysis in non-linear solid and structural mechanics. Ph.D. Thesis. Stanford, CA.: Stanford University.

George, J. A. 1971. Computer Implementation of the Finite Element Method. *Tech. Report STAN-CS-208.* Stanford, CA: Stanford University.

George, J. A., Liu, J. W. H., and Ng, E. 1987. Communication reduction in parallel sparse Cholesky factorization on a hypercube. in: *Hypercube Multiprocessors.* edited by M. T. Heath. SIAM.

George, J. A., and Liu, J. W. H. 1987. The evolution of the minimum degree ordering

algorithm. Tech. Report ORNL/TM-10452. Oak Ridge, TN: Mathematical Sciences Section, Oak Ridge National Laboratory.

George, J. A., and Liu, J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Englewood Cliffs, NJ: Prentice Hall.

George, J. A. 1973. Nested dissection of a regular finite element mesh. *SIAM J. of Numer. Anal.* 10(2): 345 - 363.

Gibbs, N. E., Poole, W. G., and Stockmeyer, P. K. 1976. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Numer. Anal.* 13(2): 236 - 250.

Hallquist, J. O. 1984. NIKE3D: An Implicit, Finite Deformation, Finite Element Code for Analyzing the Static and Dynamic Response of Three-dimensional Solids. *Tech. Report UCID-18822, Rev. 1.* Berkeley, CA: University of California.

King, S. D., and Hager, B. H. 1989. Coupling of mantle temperature anomalies and the flow pattern in the core: Interpretation based on simple convection calculations. *Phys. Earth Planet. Int.*, 58: 118-125.

King, S. D., Raefsky, A., and Hager, B. H. 1990. Conman: Vectorizing a finite element code for incompressible two-dimensional convection in the earth's mantle. *Phys. Earth Planet. Int.*, 59: 196-208.

Lucas, R. F., Blank, W. T., and Tiemann, J. J. 1987. A parallel solution method for large sparse systems of equations. *IEEE Trans. Computer Aided Design.* CAD-6(6): 981-991.

Lyzenga, G. A., Raefsky, A., and Nour-Omid, B. 1988. Implementing finite element

software on hypercube machines. In *Proceedings of the third conference on hypercube computers and applications.* Pasadena, CA: ACM Press.

Moler, C. 1987. Matrix computation on distributed memory multiprocessors. in: *Hypercube Multiprocessors*, edited by M. T. Heath. SIAM, pp. 181 - 195.

Schreiber, R. 1982. A new implementation of sparse gaussian elimination. *ACM Trans. Math. Software.* 8: 256-276.

Simon, H., Vu, P., and Yang, C. 1989. Performance of a supernodal general sparse solver on the Cray Y-MP: 1.68 GFLOPS with Autotasking. Applied Mathematics Technical Report SCR-TR-117. Seattle: Boeing Computer Services.

Travis, B. J., Anderson, C., Baumgardner, J., Gable, C., Hager, B. H., Olson, P., O'Connell, R. J., Raefsky, A., and Schubert, G. 1990. A benchmark comparison of numerical methods for infinite Prandtl number convection in two-dimendsional Cartesian geometry. *Geophys. and Astrophys. Fluid Dynamics.* : in press.

Zienkiewicz, O. C. 1977. The finite element method. London: McGraw-Hill.

Figure Captions

Figure 1 The four bilinear quadrilateral elements (1-4) all contribute to the global nodal equation at node 5, which they all share. The element numbers are circled and the global node numbers are not. The LM array for elements 1-4 are listed in Table 1.

Figure 2 The four color ordering scheme used in ConMan. Note that the shaded elements (group I) do not share nodes with any other group I element. This group can safely be operated on with a vector operation.

Figure 3 Sample psuedocode for both scalar (a) and vector (b) finite element right hand side assembly routine. The loop over the elements (numel) in (a) is replaced by a loop over BLOCK and loops over the elements within the block (iv) in (b). Short loops over the number of local nodes (nen) in (a) are unrolled in (b). numel is the number of elements, numblk is the number of element blocks, nvec is the length of the element block and nipt is the number of integration points within an element. The lm array maps elements and local nodes to global equation numbers.

Figure 4 A representation of the skyline storage scheme. X's represent a non-zero matrix element. Notice that only zeros in a column below a non-zero are stored.

Figure 5 A contrast of band, envelope and skyline storage. An initial nonzero structure is depicted in 5a. The location that would be filled in and operated upon by band, envelope, and skyline algorithms are depicted in 5b, 5c, and 5d respectively. The diagonals as well

as the initial non-zero rows and columns are highlighted.

Figure 6 An example of Nested Dissection. The dissection has been applied twice, first halving the grid with the vertical separator, then recursively halving the two disjoint subsets with the horizontal separators. The resulting matrix structure is also depicted. The empty areas are all logically zero, the dark areas correspond to dense submatrices, and the gray areas represent the four quadrants of the grid that have not yet been ordered.

Figure 7 A 5 by 5 grid and the elimination tree created by a Nested Dissection ordering.

Figure 8 a) A sample sparse matrix derived from a 5 by 5 grid. The matrix is permuted to reflect the same ordering as used in Figure 7. The original equation numbers are displayed to the left of each row. The column permutations are the same as the row permutations. b) The front after initialization with equations 3 and 8. c) The real stack containing the update matrices from equations 7 and 9. d) The front after the addition of update matrices from equations 7 and 9. e) The front after the elimination of equations 3 and 8.

Figure 9 A graphical representation of the total run times (in seconds) for the four benchmark problems. The numbers above each group represent the ratio of the scalar skyline execution time to the vector DMF execution time.

Figure 10 The execution times (in seconds) for the finite element routines from BM1, BM2 and BM3 in both scalar and vector modes on the Cray X-MP/18. Notice the

dramatic increase with grid size for the scalar case.

Figure 11 A breakdown of the execution times (in seconds) for the vectorized finite element routines shown in Figure 10. The gather/scatter time is mostly due to scattering the element stiffness matrices into the global stiffness matrix.

Figure 12 The execution times (in seconds) for the vectorized skyline factor and the vectorized DMF factor for BM1, BM2 and BM3 on the Cray X-MP/18. Notice the dramatic increase with grid size for the skyline case.

Table Captions

Table 1 A representation of the *lm* array for Figure 1. The *lm* array is a two dimensional array which has dimensions number-of-elements by number-of-local-nodes. The value of the *lm* entry is the global equation number. For element 2 local node 1 the global equation number is 4.

Table 2 Data for the four benchmark problems. The average bandwidth is defined to be the maximum $|i-j|$ where A(i,j) is not equal to zero. Time steps is the number of times the matrix is factored (for BM3 and BM4 scalar runs were only run 50 time steps and timing results were multiplied by 10). The memory size is in Cray megawords (where one floating point number takes one word) and is taken from the Cray hardware performance monitor.

Table 3 Timing data (in seconds) from Cray X-MP/18 for the four benchmark problems. Times are reported for both scalar and vector runs using the skyline solver but only vector for the DMF solver. Compiler directives were used to force vectorization where ever necessary but no assembly language kernels or library routines were used.

Biographies

Scott King is a Green Scholar at the Institute of Geophysics and Planetary Physics, Scripps Institute of Oceanography, UCSD, La Jolla. He received a B.A. in geological sciences and applied mathematics from the University of Chicago in 1985; and a Ph.D. in geophysics from California Institute of Technology, Pasadena, in 1990. While at Caltech, he developed a vectorized finite element code for studying thermal convection in the Earth's mantle. His research interests include geophysical fluid dynamics, parallel computing, and numerical fluid dynamics.

Robert Lucas is currently a member of the research staff at the Supercomputing Research Center in Bowie, MD. He received his Ph.D. in electrical engineering from Stanford in 1988. He also received his M.S. (1983) and B.S. (1980) in electrical engineering from Stanford. He worked for Hughes Aircraft Co. Ground Systems Group on naval tactical displays from 1979 until 1983, with General Electric in 1984, and Intel Scientific Computers in 1986. His research interests include numerical linear algebra and parallel processing.

Arthur Raefsky currently working for Centric Engineering Systems Inc. developing an

graphically driven interactive finite element analysis package. He designed numerical algorithms and software for Caltech's Mark II and III Hypercube as a member of the technical staff at Jet Propulsion Laboratory, Pasadena, CA and Caltech Concurrent Computing Project (C3P), California Institute of Technology, Pasadena and the Electrical Engineering Department at Stanford. This work received the 1986 NASA award for technical innovation. He developed the educational software package included in the book *The Finite Element Method* by T. J. R. Hughes, Prentice Hall, 1987. He has also worked on numerical algorithms and software for the Cray X/MP, Cray 2, and Convex C1 computers. He received his Ph.D. in Applied Mathematics and Computer Science from State University of New York at Binghamton in 1977; M.S. in applied mathematics and computer science from State University of New York at Binghamton in 1974; and B.S. in physics from New York Institute of Technology in 1971. His interests include parallel computing, algorithm design and implementation, and visualization.

LM Array for the grid in Figure 1

Element	Local Node 1	2	3	4
1	1	4	5	2
2	4	7	8	5
3	5	8	9	6
4	2	5	6	3

Table 1

Data for ConMan Benchmarks

Benchmark	BM1	BM2	BM3	BM4
Number of Elements in the X direction	32	64	96	190
Number of Elements in the Z direction	32	64	96	48
Boundary Conditions	fs	fs	fs	periodic
Number of Active Equations	2046	8190	18430	18238
Average Bandwidth	65	129	193	190†
Time steps	500	500	500‡	500‡
Total memory skyline (MW)	0.13	1.06	3.6	3.5
Total memory dmf (MW)	0.32	1.4	3.2	3.3
Millions of Operations (factor skyline)	13	204	1,000	1,000
Millions of Operations (factor dmf)	9.7	90	298	352

†estimated
‡dmf 500 steps, scalar 50 steps × 10

Table2

ConMan Benchmarks

all times in seconds from Cray X-MP/18

Benchmark	routine	scalar skyline	vector skyline	vector dmf
BM1	f_vstf:	111.2	18.0	13.7
	f_vres:	3.6	0.7	0.7
	f_tres:	38.3	4.0	4.0
	factor:	536.8	357.8	47.3
	back-slv:	21.0	9.0	8.8
	symbolic:	—	—	0.4
	total:	722.6	392.0	77.4
BM2	f_vstf:	462.7	75.5	55.7
	f_vres:	14.8	2.7	2.7
	f_tres:	152.2	16.3	15.5
	factor:	6270.0	3042.3	296.0
	back-slv:	148.9	37.6	38.6
	symbolic:	—	—	1.6
	total:	7122.4	3184.6	420.0
BM3	f_vstf:	1247.0	176.4	128.2
	f_vres:	40.0	6.3	6.3
	f_tres:	341.0	36.5	36.5
	factor:	32383.0	10643.3	843.2
	back-slv:	561.0	98.7	91.3
	symbolic:	—	—	3.6
	total:	35350.0	10984.0	1129.7
BM4	f_vstf:	1245.0	132.4	87.9
	f_vres:	39.0	4.0	4.0
	f_tres:	342.0	36.1	36.1
	factor:	3089 .0	10787.9	915.3
	back-slv:	555.0	97.2	91.5
	symbolic:	—	—	3.7
	total:	33854.0	11080.3	1161.0

Table 3

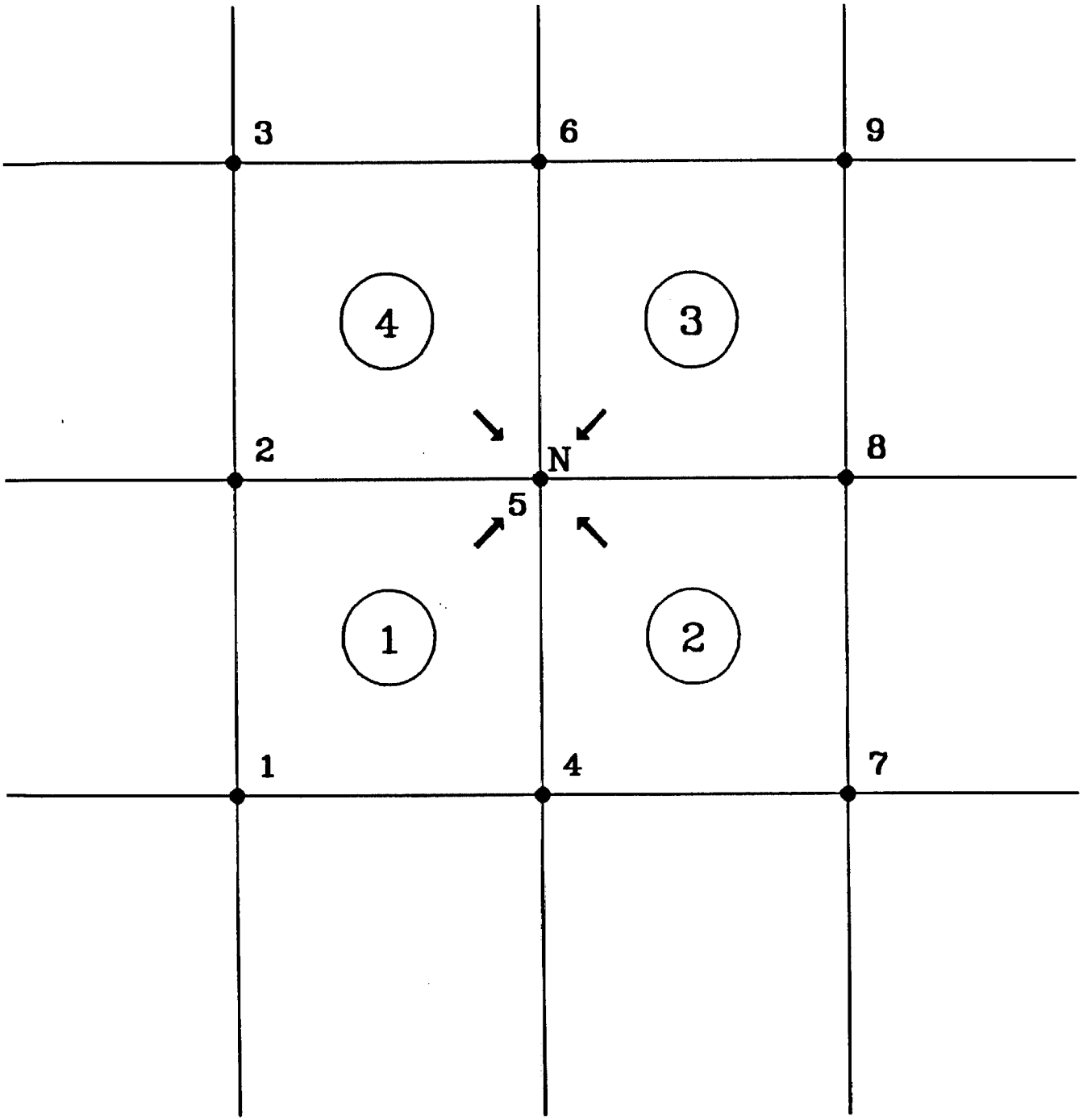


Figure 1

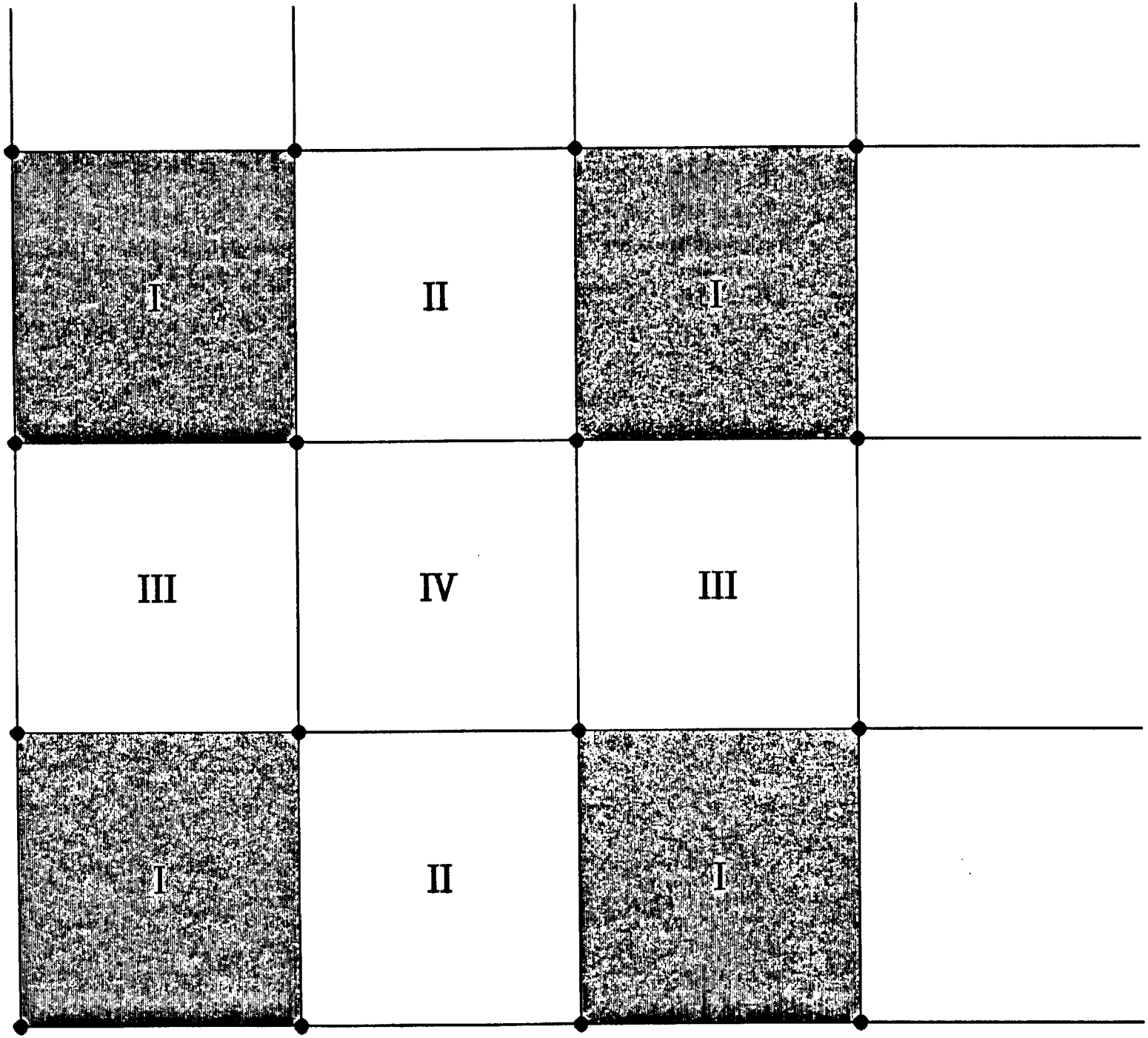


Figure 2

Scalar Pseudocode

```
do e = 1, numel
    do n = 1, nen
        data_local(n) ← data( lm(e,n) )
    enddo
    do L = 1, nipt
        do n = 1, nen
            k_local(n) ← k_local(n) +
            data_local(n)*shl(n,L)*det(L)
        enddo
    enddo
    do n=1,nen
        k(lm(e,n)) ← k(lm(e,n)) + k_local(n)
    enddo
enddo
```

Vector Pseudocode

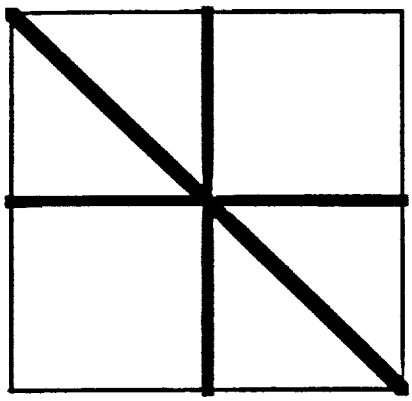
```
do block =1,numblk
    do iv = 1, nvec
        data_local(iv,1) ← data(lm(ivel,1))
        data_local(iv,2) ← data(lm(ivel,2))
        data_local(iv,3) ← data(lm(ivel,3))
        data_local(iv,4) ← data(lm(ivel,4))
    enddo
    do L = 1, nipt
        do iv = 1, nvec
            k_local(iv,1) ← k_local(iv,1) +
                data_local(iv,1)*shl(1,L)*det(iv,L)
            k_local(iv,2) ← k_local(iv,2) +
                data_local(iv,2)*shl(1,L)*det(iv,L)
            k_local(iv,3) ← k_local(iv,3) +
                data_local(iv,3)*shl(1,L)*det(iv,L)
            k_local(iv,4) ← k_local(iv,4) +
                data_local(iv,4)*shl(1,L)*det(iv,L)
        enddo
    enddo
    do iv = 1, nvec
        k(lm(ivel,1)) ← k(lm(ivel,1)) + k_local(iv,1)
        k(lm(ivel,2)) ← k(lm(ivel,2)) + k_local(iv,2)
        k(lm(ivel,3)) ← k(lm(ivel,3)) + k_local(iv,3)
        k(lm(ivel,4)) ← k(lm(ivel,4)) + k_local(iv,4)
    enddo
enddo
```

Figure 3

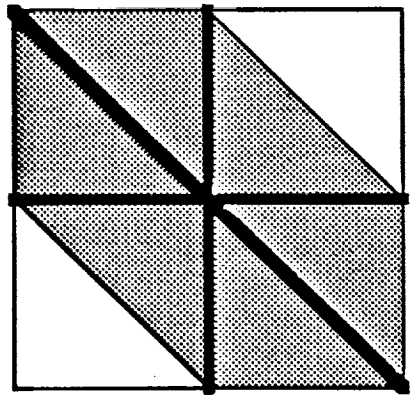
Skyline Storage

$$[K] = \begin{pmatrix} X & X & 0 & 0 & 0 & 0 & 0 & X \\ & X & X & 0 & X & 0 & 0 & 0 \\ & & X & 0 & X & 0 & 0 & 0 \\ & & & X & 0 & X & 0 & X \\ & & & & X & X & 0 & 0 \\ & & & & & X & X & 0 \\ & & & & & & X & X \\ & & & & & & & X \end{pmatrix}$$

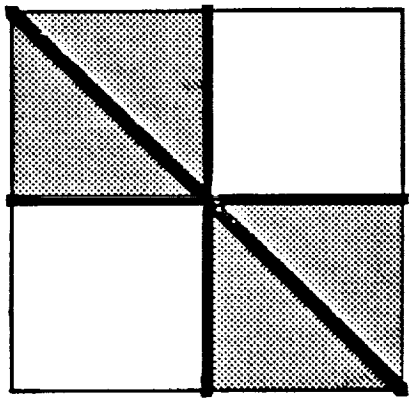
Figure 4



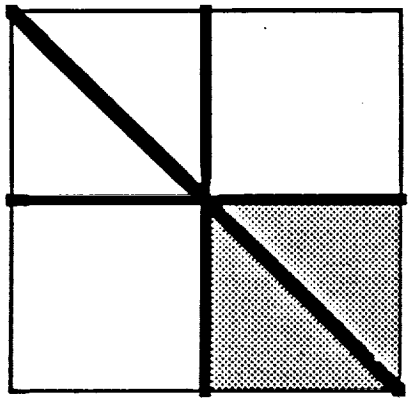
Initial Nonzero Structure



Banded Nonzero Structure

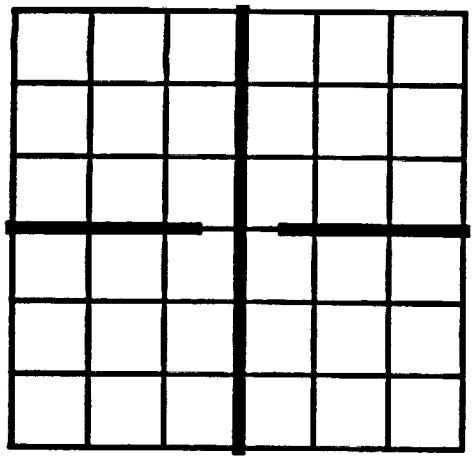


Envelope Nonzero Structure



Skyline Nonzero Structure

Figure 5



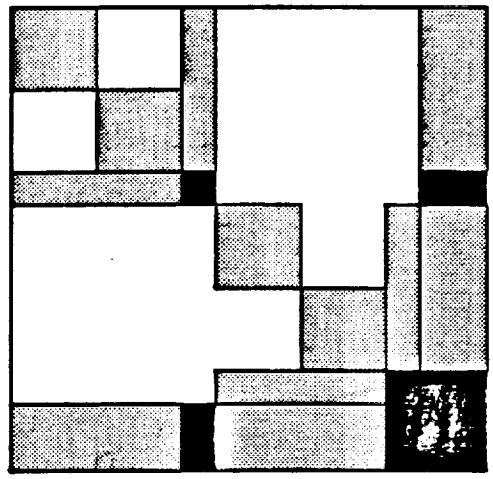
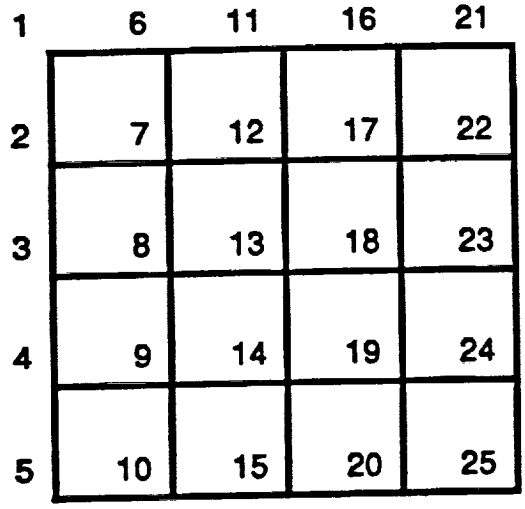


Figure 6


1 6 11 16 21
2 7 12 17 22
3 8 13 18 23
4 9 14 19 24
5 10 15 20 25

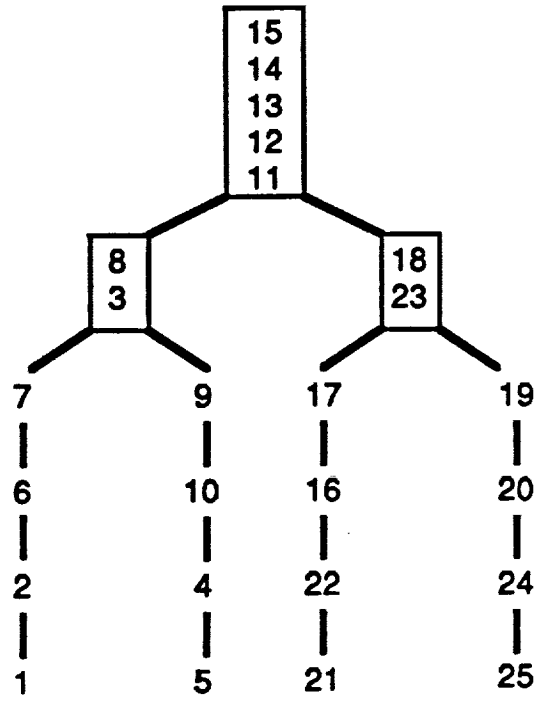

15
14
13
12
11
8
3
18
23
7
9
17
19
6
10
16
20
2
4
22
24
1
5
21
25

Figure 7

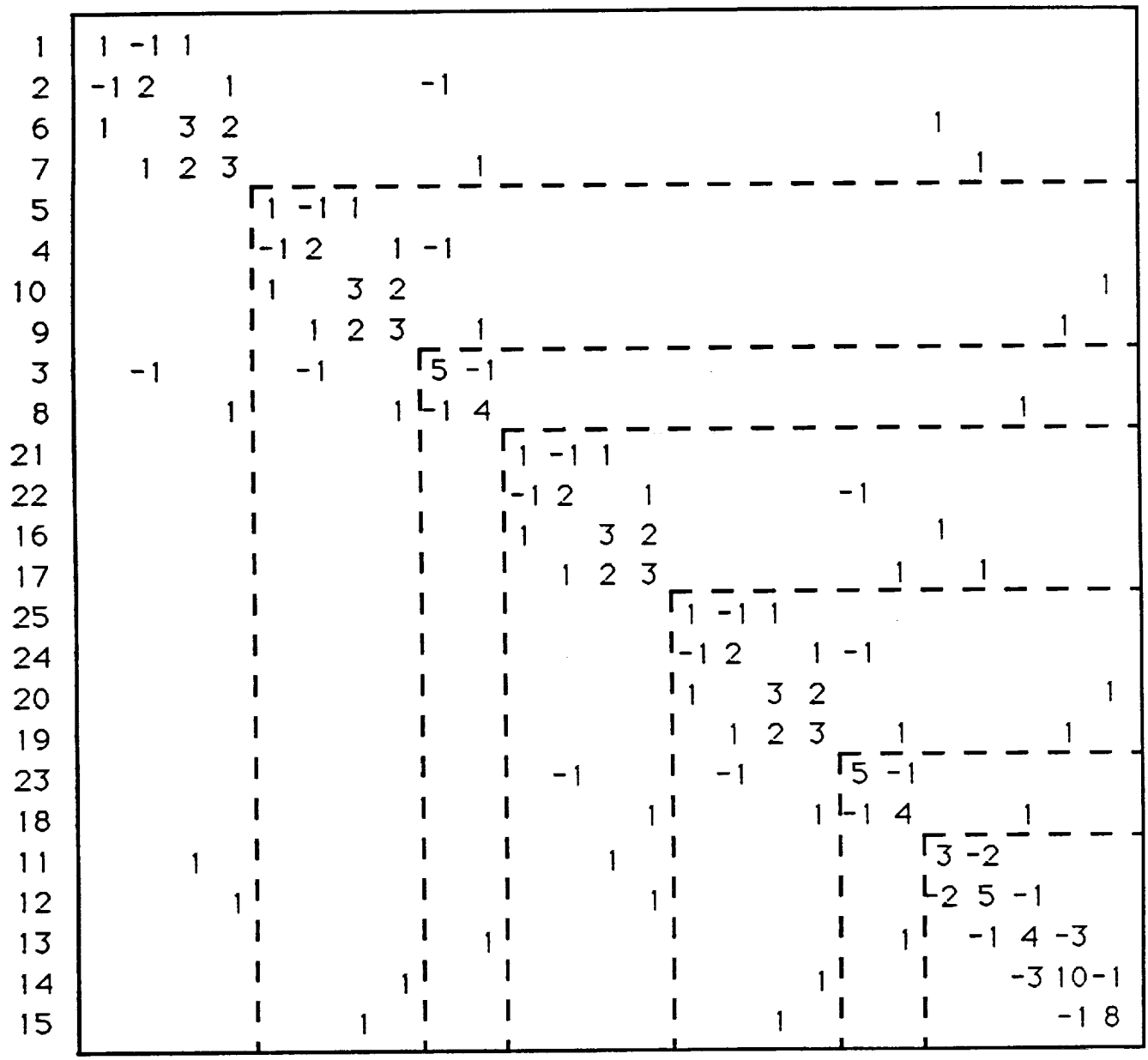


Figure 8a
Sample sparse matrix derived from 5 by 5 grid
and 5 point stencil. Matrix is permuted to reflect
the same ordering as used in Figure 7.

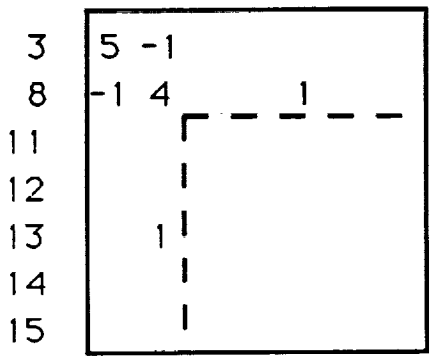


Figure 8b
Front after initialization
with equations 3 and 8.

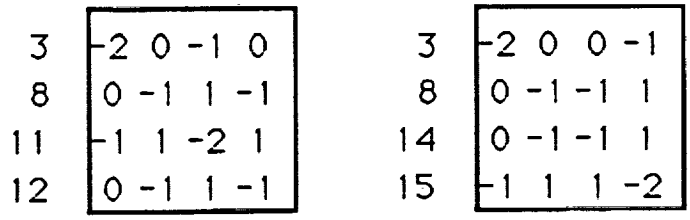


Figure 8c
Real Stack containing update matrices
from equations 7 and 9.

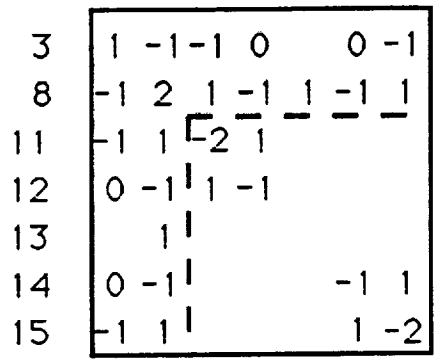


Figure 8d

Front after addition of update matrices from equations 7 and 9.

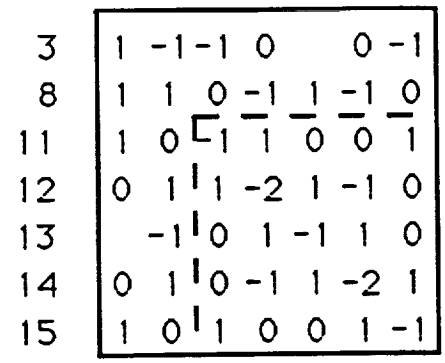


Figure 8e

Front after elimination of equations 3 and 8.

Total Run Times: Cray X-MP/18

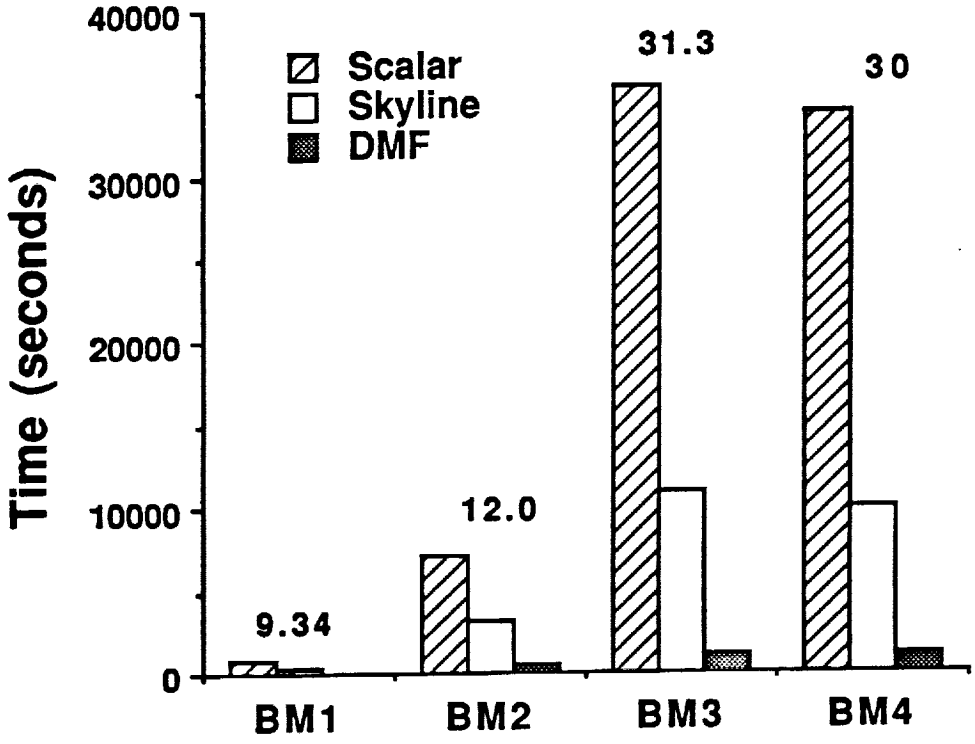


Figure 9

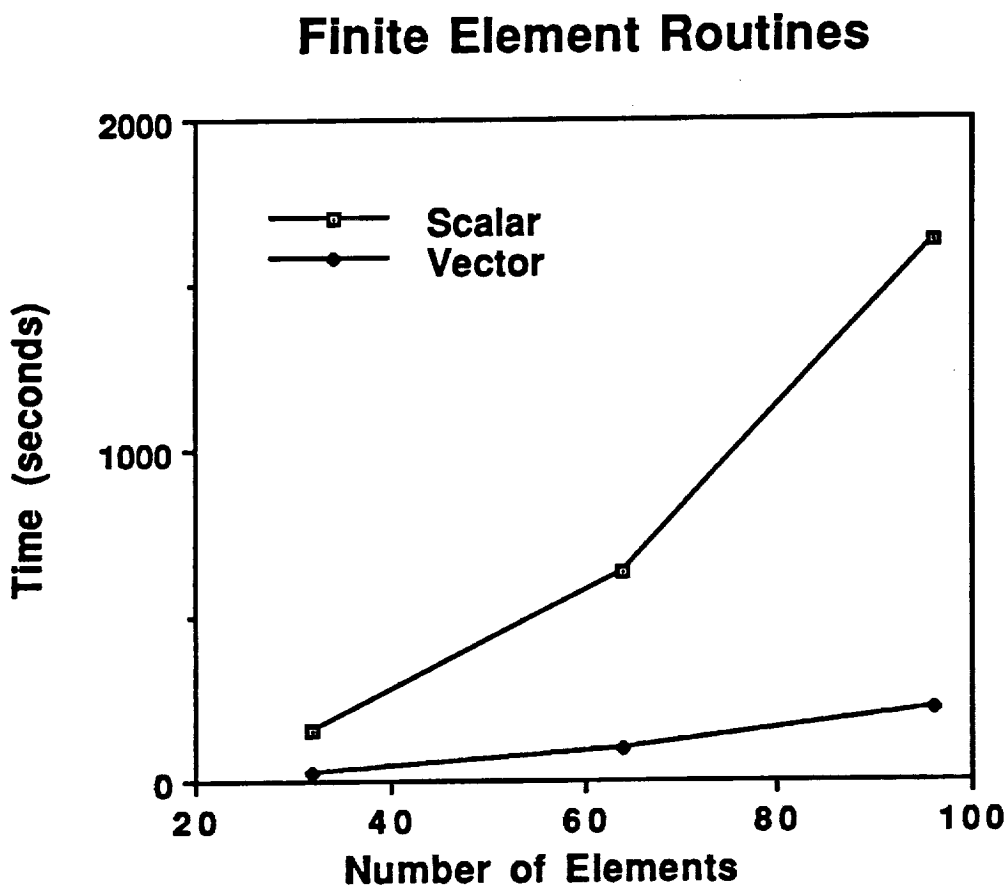

Finite Element Routines
2000
1000
0
Time (seconds)
Scalar
Vector
20
40
60
80
100
Number of Elements

Figure 10

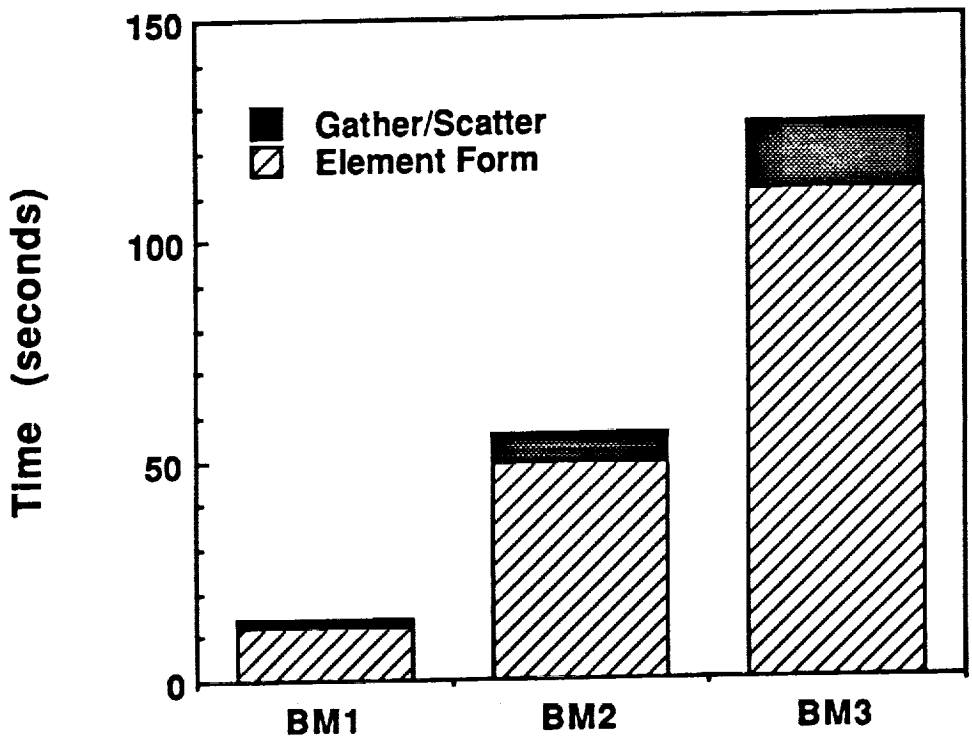

150
100
50
0
Time (seconds)
Gather/Scatter
Element Form
BM1
BM2
BM3

Figure 11

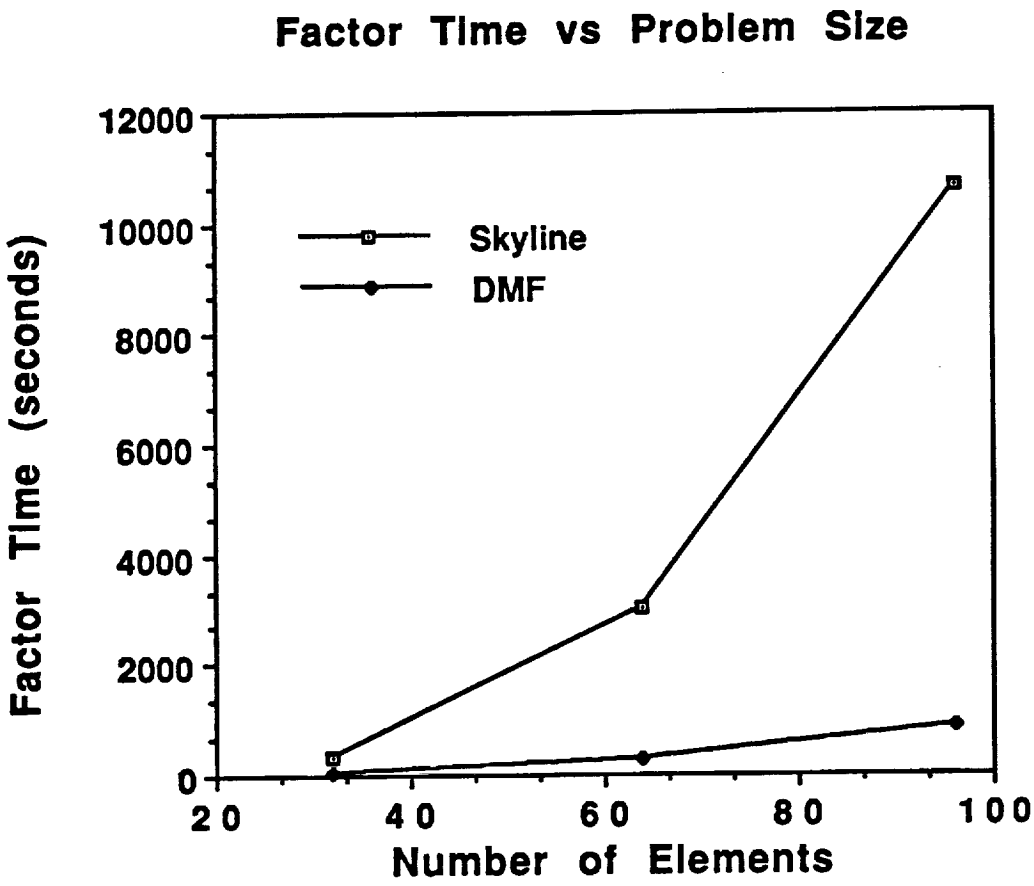

Factor Time vs Problem Size
Factor Time (seconds)
12000
10000
8000
6000
4000
2000
0
Skyline
DMF
20
40
60
80
100
Number of Elements

Figure 12